CSCI 740 - Programming Language Theory

Lecture 21

Verification Condition Generation (I)

Instructor: Hossein Hojjat

October 23, 2017

Hoare Rules: Summary

$$\frac{}{\vdash \{A[x \mapsto e]\}\ x := e\ \{A\}} \qquad \frac{\vdash \{A \land b\}\ c_1\ \{B\} \qquad \vdash \{A \land \neg b\}\ c_2\ \{B\}}{\vdash \{A\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{B\}}$$

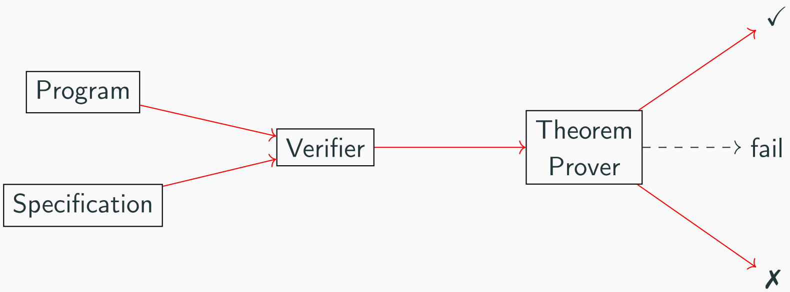
$$\frac{\vdash \{A \land b\}\ c\ \{A\}}{\vdash\ \{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{A \land \neg b\}} \qquad \frac{\vdash \{A\}\ c_1\ \{C\} \qquad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1\ ;\ c_2\ \{B\}}$$

$$\frac{\vdash A' \Rightarrow A \quad \vdash \{A\}\ c\ \{B\} \quad \vdash\ B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

Automating Reasoning in Hoare Logic

- Manually proving correctness is tedious
- We'd like to automate the tedious parts of program verification
- Idea: Assume an oracle gives loop invariants - we can then automate the rest of the reasoning
- This oracle can either be a human or a static analysis tool
    - (e.g., abstract interpretation)

Automated Verification



- Example specs: safety (no crashes), absence of arithmetic overflow, complex behavioral property (e.g., “sorts an array”)

Verification Conditions (VC)

- Automating Hoare logic is based on generating verification conditions (VC)
- A verification condition is a formula ϕ such that program is correct iff ϕ is valid
- Deductive verification has two components:

1. Generate VC's from source code
2. Use theorem prover to check validity of formulas from step 1

Generating VCs: Forwards vs. Backwards

$$\{A\} \quad \text{program} \quad \{B\}$$

Precondition   Postcondition

- Two ways to generate verification conditions: forwards or backwards
- A forwards analysis starts from precondition and generates formulas to prove postcondition
- Forwards technique computes strongest postconditions (sp)
- In contrast, backwards analysis starts from postcondition and tries to prove precondition
- Backwards technique computes weakest preconditions (wp)

Strongest Postconditions

- Some valid Hoare Triples

$$\begin{array}{lll} \{x = 5\} & x := x + 5 & \{\mathsf{true}\} \\ \{x = 5\} & x := x + 5 & \{x > 0\} \\ \{x = 5\} & x := x + 5 & \{x = 10 \lor x = 5\} \\ \{x = 5\} & x := x + 5 & \{x = 10\} \end{array}$$

- All are valid but $x = 10$ is the most useful one
  - Strongest postcondition
- If $\{P\}\ S\ \{Q\}$ and for all Q' such that $\{P\}\ S\ \{Q'\}$, $Q \Rightarrow Q'$, then Q is the strongest postcondition of S with respect to P
- check: $x = 10 \Rightarrow \mathsf{true}$
- check: $x = 10 \Rightarrow x > 0$
- check: $x = 10 \Rightarrow x = 10 \lor x = 5$
- check: $x = 10 \Rightarrow x = 10$

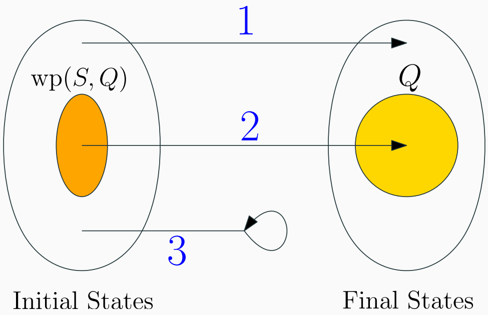
Weakest Preconditions

- Some valid Hoare Triples (assume an extension of IMP with division)

$$
\begin{array}{lll}
\{x = 5 \land y = 10\} & z := x/y & \{z < 1\} \\
\{x < y \land y > 0\} & z := x/y & \{z < 1\} \\
\{y \neq 0 \land x/y < 1\} & z := x/y & \{z < 1\}
\end{array}
$$

- All are valid but $y \neq 0 \land x/y < 1$ is the most useful one
- It allows us to invoke the program in the most general condition
    - Weakest precondition
- If $\{P\}\ S\ \{Q\}$ and for all P' such that $\{P'\}\ S\ \{Q\}$, $P' \Rightarrow P$, then P is the weakest precondition $\mathrm{wp}(S, Q)$ of S with respect to Q

Weakest Preconditions



- Arc 1 produces a final state not satisfying Q ✗
- Arc 2 produces a final state satisfying Q ✓
- Arc 3 gets into a loop and produces no final state ✗

Backwards Method

- Idea: Suppose we want to verify Hoare triple $\{P\}\ S\ \{Q\}$
- Start with Q and go backwards, compute formula $\mathsf{wp}(S, Q)$
- We can dually define forward method using sp
  - (won't discuss in lecture)
- Weakest preconditions are defined inductively following Hoare's rules

- $\mathsf{wp}(x := E, Q) = Q[x \mapsto E]$
- $\mathsf{wp}(c_1; c_2, Q) = \mathsf{wp}(c_1, wp(c_2, Q))$
- $\mathsf{wp}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, Q) = b \to \mathsf{wp}(c_1, Q) \wedge \neg b \to \mathsf{wp}(c_2, Q)$

- Rule for conditional says:
  "If b holds, wp of then branch must hold; otherwise,
  wp of else branch must hold"

Exercise

- Consider the following code S

$$x := y + 1;\ \texttt{if}\ x > 0\ \texttt{then}\ z := 1\ \texttt{else}\ z := -1$$

- What is $\mathsf{wp}(S, z > 0)$?
- What is $\mathsf{wp}(S, z \leq 0)$?

- $\mathsf{wp}(x := E, Q) = Q[x \mapsto E]$
- $\mathsf{wp}(c_1; c_2, Q) = \mathsf{wp}(c_1, wp(c_2, Q))$
- $\mathsf{wp}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, Q) = b \to \mathsf{wp}(c_1, Q) \land \neg b \to \mathsf{wp}(c_2, Q)$

Exercise

- Consider the following code S

$$x := y + 1;\ \texttt{if}\ x > 0\ \texttt{then}\ z := 1\ \texttt{else}\ z := -1$$

- What is $\mathsf{wp}(S, z > 0)$? $y \geq 0$
- What is $\mathsf{wp}(S, z \leq 0)$?

- $\mathsf{wp}(x := E, Q) = Q[x \mapsto E]$
- $\mathsf{wp}(c_1; c_2, Q) = \mathsf{wp}(c_1, wp(c_2, Q))$
- $\mathsf{wp}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, Q) = b \rightarrow \mathsf{wp}(c_1, Q) \wedge \neg b \rightarrow \mathsf{wp}(c_2, Q)$

Exercise

- Consider the following code S

$$x := y + 1;\ \texttt{if}\ x > 0\ \texttt{then}\ z := 1\ \texttt{else}\ z := -1$$

- What is $\mathsf{wp}(S, z > 0)$? $y \geq 0$
- What is $\mathsf{wp}(S, z \leq 0)$? $y < 0$

- $\mathsf{wp}(x := E, Q) = Q[x \mapsto E]$
- $\mathsf{wp}(c_1; c_2, Q) = \mathsf{wp}(c_1, wp(c_2, Q))$
- $\mathsf{wp}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, Q) = b \rightarrow \mathsf{wp}(c_1, Q) \wedge \neg b \rightarrow \mathsf{wp}(c_2, Q)$

Assert, Assume and Havoc

- It is convenient to extend the language with the following statements:
- `havoc` x — change x to an arbitrary value
- `assert` b — if b holds, terminate; otherwise, exit with error
- `assume` b — if b holds, terminate; otherwise, block

```
{ x=x0 and y=y0 }
z = x;
x = y;
y = z;
{ y=x0 and x=y0 }
```

$\Longrightarrow$

```
assume x = x0;
assume y = y0;
z = x;
x = y;
y = z;
assert x = x0;
assert y = y0;
```

Assert, Assume and Havoc

- wp($\texttt{assert}\ b, Q) = b \land Q$
- For Q to be true after, b must also be true before, because otherwise we won't get past the assert
- wp($\texttt{assume}\ b, Q) = b \rightarrow Q$
- If b is not true, we don't care if Q is satisfied
- wp($\texttt{havoc}\ x, Q) = \forall y. Q[x \mapsto y]$

Weakest Preconditions for Loops

- Let's start from the equivalence
  $\texttt{while}\ b\ \texttt{do}\ c = \texttt{if}\ b\ \texttt{then}\ (c; \texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else skip}$
- Assume $W = \mathsf{wp}(\texttt{while}\ b\ \texttt{do}\ c, Q)$
- It must be that: $W = (b \Rightarrow \mathsf{wp}(c, W) \wedge \neg b \Rightarrow Q)$
- But this is a recursive equation!

Weakest Preconditions for Loops

- Can be computed by using an invariant

$$\mathsf{wp}(\texttt{while}\ b\ \texttt{do}\ c, Q) =$$
$$I \wedge \forall \vec{y}. \Big((I \wedge b \Rightarrow \mathsf{wp}(c, I)) \wedge (I \wedge \neg b \Rightarrow Q) \Big) [\vec{x} \mapsto \vec{y}]$$

- where $\vec{x} = x_1, \cdots, x_n$ are variables modified in c and I is the loop invariant

Intuition

- We encode the meaning of `while` statement as following:

```
while b do c ≡
 assert I;
 havoc x_1, ..., x_n;
 assume I;
 if (b) then
 {
  c
  assert I;
  assume false;
 }
```

- `assert` I: Loop invariant is checked on entry
- We "fast forwards" to an arbitrary iteration by setting the variables to arbitrary values
- In that arbitrary iteration we check that the loop condition is defined
- Either we perform one more iteration or terminate the loop
- `assume false` indicates that the remainder of program is not analyzed immediately after an arbitrary loop iteration
- Analysis only proceeds under the successful termination